Aubonnet *et al. Journal of Internet Services and Applications* (2015) 6:15
DOI 10.1186/s13174-015-0031-7

Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

Management of service composition based on self-controlled components

Tatiana Aubonnet[1*†], Ludovic Henrio[2†], Soumia Kessal[3†], Oleksandra Kulankhina[4†], Frédéric Lemoine[1†], Eric Madelaine[4†], Cristian Ruz[5†] and Noëmie Simoni[3†]

Abstract

Cloud computing and Future Internet promise a new ecosystem where everything is "as a service", reachable and connectable anywhere and anytime, everyone succeeding to get a service composition that meets his needs. But do we have the structure and the appropriate properties to design the service components and do we have the means to manage, at run-time, the personalised compositions corresponding to Service Level Agreement? In this article we introduce an entity of service composition called Self-Controlled Component (SCC), including, since the design step, functional and non-functional specifications. SCCs benefit both from the strong structure, explicit composition, and autonomic management of component-oriented programming, from the highly dynamic composition, and from the discovery capacities of service-oriented computing. Self-control mechanisms are then attached automatically to SCCs to enable autonomic application management during execution. The objective of this new concept is to provide strong Quality of Service (QoS) guarantees of composed applications. We illustrate the approach using an example called Springoo, to how in the context of a legacy application the contributions and benefits of our solution. For the management of the service composition we propose the concept of Virtual Private Service Network (VPSN) and Virtual Service Community (VSC) that allows us to model the personalised Service Level Agreement (SLA) where user requirements and provider offers converge on a QoS contract.

Keywords: Service component; Quality of service; Autonomic control; Service composition

1 Introduction

Cloud computing and Future Internet ecosystems are attractive for several reasons. They allow designing your own application from components offered in a catalog, and connecting almost everything. This approach enables the nearly automatic creation of solutions corresponding to the user's demand.

In addition, one of the slogans of cloud computing is "to pay-as-you-go". It means that the supplier is able to adapt to the user's needs. His ability to manage the elasticity, the high availability, and the on-demand provisioning is a part of his offer. The integration, from the design phase, of automated management procedures is always desirable, but it is even more crucial in the competitive context of this new ecosystem.

From the cloud provider point of view, the objective is to meet the required properties based on customer requirements and needs. In practice, many cloud providers offer the same services that differ in their quality of service levels, price, and in the way they are created, deployed, and managed.

As a consequence, the request and the offer must be entirely and explicitly documented and guaranteed by the Service Level Agreement (SLA).

The questions are: (i) how to adapt the component models in order to ensure the convergence of supply and demand? (ii) how to ensure both the autonomy of each component and the end-to-end quality of the service? and (iii) what to record into the SLA to ensure its practical usability and its veracity?

(i) For converging supply and demand, one would have to choose components "as a Service", the user must pick the services corresponding to the behaviour and the quality of service he expects. Moreover, the service composition should facilitate their adaptation on demand,

*Correspondence: tatiana.aubonnet@cnam.fr
†Equal contributors
[1]CNAM (Conservatoire National des Arts et Métiers), Computer Science Department, 292 rue Saint Martin, 75003 Paris, France
Full list of author information is available at the end of the article

it should reduce the functional dependencies between services, and have loose bindings between entities of the composition as Service Oriented Architecture (SOA) advocates. Indeed, without these loose ties the composition cannot be customized and the interconnection of components cannot vary on demand.

(ii) Concerning guarantees, we need the right information at the right place in order to take the desired decisions. It is not enough for the service composition to be agile at functional level, but it must also be agile and dynamic at management level, i.e. at the non-functional level. In more details, the management and the SLA guarantees should be made hierarchical and distributed among services; part of the adaptation decisions should be taken locally, but other parts must be taken from a global point of view. The structured composition and service management we present in this paper allows for a precise control of the place where decisions are taken and enacted. This enables dynamic and precise adaptation of service composition by providing the means to react at different levels when the service does not fulfill its contract.

(iii) Concerning the SLA definition, requirements, resources provided and penalties for breach of contract are to be recorded. It must be expressed in a vocabulary understandable by different parties. While the service is expressed when the service is made available, it is difficult to ensure the exactitude of the SLA, but our approach allows us to monitor the service out of contract and exclude them from the compositions.

This paper provides a composition framework that transforms the user-defined choices (demand) into the right services (offer) and that adapts automatically this composition at run-time. The approach we advocate allows a personalized and automated composition of services with service level guarantees accordance with SLA. Our main contributions are the following:

- We define service components that at the same time provide the guarantee of a certain service level and enable autonomic adaptation of the composition to ensure that this service quality will be guaranteed during the application execution.
- We provide an architecture for composing services featuring service discovery and SLA-based adaptation.
- We design generic Quality of Service (QoS) components guaranteeing that the service composition will provide the predefined functionality and QoS. This is realised by the definition of a Virtual Private Service Network (VPSN) defining the user's application and then, based on this description, choosing and replacing at runtime the services involved in the composition.
- Those contributions are provided in a programming and execution environment that offers ease of programming, location and distribution transparency, and autonomic adaptation.

In this paper we analyse the related works in Section 2 and propose an approach to compose components "as services" which can be self-controlled; such components are called Self-Controlled service Component (SCC) and presented in Section 3. Our work uses a design platform with support for correct composition of components, presented in Section 4; this platform provides tools to instantiate components that have to be adapted to provide controlled service components; this adaptation process is described in Section 5. A solution for service composition and its adaptation is then proposed (Section 6) and a use-case named Springoo (Section 7) provides an integration example. The Section 8 offers Service Level Objective (SLO) and SLA based on SCC, and Section 9 presents experimental results. Finally a conclusion (Section 10) ends the paper.

2 Related work

Analysis of existing solutions to meet the needs of distributed systems and cloud has led us to look at the evolution leading from component models (Fractal, Grid Component Model (GCM)) to service models SOA (Section 2.1), to their management (Section 2.2) and thus to position our work (Section 2.3).

2.1 From component models to service models

Component models provide a structured programming paradigm, and ensure a very good re-usability of programs. In component applications, dependencies are defined together with provided functionalities by the means of provided/required ports; this improves the program specification and thus its re-usability. We focus here mostly on hierarchical component models because they make the design of large-scale systems easier. A component model is said to be hierarchical if the composition of several components is also a component that can be used at a higher composition level. We call *primitive components* the leaves of the composition tree, i.e. the components that contain the business code.

Fractal [1] is a general component model, which is intended to implement, deploy and manage complex software systems. It showed its effectiveness in the particular setting of operating systems and middleware, through the use of interfaces (usage, control and management). But if the functional interconnections are explicit, those of management are less obvious.

The Grid Component Model (GCM) [2], is an extension of Fractal targeting specifically distributed systems. A strong point of GCM is the separation of concerns [3].

In GCM, the membrane, i.e. the management part of the component, can be defined precisely with all necessary interconnections between management features and with the rest of the component hierarchy. GCM/ProActive is the reference implementation of the GCM component model. It is based on the ProActive Java library [4].

SOA promotes a different composition pattern based on loosely coupled services; which is a crucial property for a personalized service session. Service Component Architecture (SCA) [5] and WS (Web Services) are major approaches supporting SOA.

SCA is a component model adapted to Service Oriented Architectures. It enables modeling service composition and creation of service components. Numerous platforms implement the SCA model, like Tuscany, Newton, or FraSCAti. The main SCA properties are: interconnection, autonomy, loose coupling, and reuse. However, the SCA model is focused on static description of components and does not standardise the runtime evolution of applications.

WS [6] is the technology supporting the flexible composition [7], the "Mashup" [8, 9] and the methods of integration services in Cloud Computing [10–12]. WS allows the implementation of SOA recommendations, it supports Web Services Description Language (WSDL) as a description language, communication APIs like Simple Object Access Protocol (SOAP) and Representational State Transfer (REST), and coordination of services through Business Process Execution Language (BPEL) [13]. Though quite exhaustive, none of these technologies offers strong support for non-functional features like non-functional management and quality of service. Service coupling is also too tight to allow true dynamic service evolution: WS offers no particular support for the replacement of a service by another one at the middle of the execution of the application.

Unfortunately, the loose coupling of WS come at some price: there is no way to control efficiently service composition because there is no tool allowing an application manager to introspect and modify service dependencies. The explicit structure of components is more restrictive but enables this control.

2.2 Autonomic management

In distributed systems and especially in the Cloud, the ability to modify at run-time execution parameters but also the architecture of the application paves the way for the autonomic adaptation of applications. In distributed systems in particular, dynamic adaptation is even more important, as the structure of components can also be used at run-time to discover services and use the most efficient service available.

Some component models and their implementations keep a trace at run-time of the component structure and their dependencies. Knowing how components are composed and being able to modify this composition at run-time provides great adaptation capabilities: the application can be adapted to evolutions in the execution environment, by changing some of the components taking part in the composition or changing the dependencies between the involved components. We call reconfiguration the actions consisting in changing at run-time the component structure, by adding or removing components in the system or by changing the way components are bound together. FraSCAti [14] is an implementation of the SCA model built upon Fractal, somehow close to GCM. It provides dynamic reconfiguration of SCA component assemblies, a binding factory, a transaction service, and a deployment engine of autonomous SCA architecture.

Additionally, if components are structured in a hierarchical manner, the adaptation can be realised in a modular manner, where each (service) component is responsible for its own adaptation and for its own quality of service; while interacting with its sub-components, or its external services for distributing the adaptation process. A deeper study of the interplay between hierarchical component models and their reconfiguration can be found in [15] which illustrates the relation between reconfiguration and hierarchy in the context of the SOFA2.0 component model.

Autonomic adaptation rules are often expressed as event condition action rules like in Automate [16] or Safran [17]. More generally, the adaptation procedure can be structured as a Monitor-Analyse-Planning-Execute (MAPE) loop for autonomic computing [18, 19]. GCM provides a framework for structuring the elements of the MAPE loop (Monitoring, Analysis, Planning, Execution) as components embedded with the management part of the components.

However those approach relies on a tight coupling between components, and components must exactly correspond to one physical entity. In service oriented architectures that can be cross-organizational, and that rely on service discovery and interchangeability, some additional effort has to be made to integrate the autonomy of each service into autonomic loosely coupled applications. In this paper we show how to address this challenge and guarantee the QoS requested by the user when choosing the service. This will be done by an approach that integrates in a single entity the notion of component and the notion of interchangeable services.

2.3 Positioning

On one side SLA and contracts for services already exist [20–22] but they lack the compositional design and management featured by components. On the other side components are very compositional and expressive, but the support for SLA, in the specific context

of service-oriented components, is very weak. Our proposal presented in this paper is to use GCM and its strongly structured entities to provide a service oriented component platform that eases the design and execution of self-controlled services with SLA guarantees. In other words, our contribution is first to define service-oriented GCM components, i.e. hierarchical components restrained to service-oriented features. Relying on this model, we design specific support for SLA and contracts dedicated to these service-components. In the following We will describe our notion of SCC components that is components that can be integrated in a service-oriented approach with dedicated management features. We retain the GCM model for its membrane that comes with a high-level framework for the development of management capabilities [3]. For dynamic management, it will be performed modularly, where each service (component) is responsible for its own adaptation and its own quality of service; while interacting with internal or external services. A strong point of our proposal is to link the provider and the consumer of the service through a SCC based SLA.

3 Self-controlled service component

Cloud computing and the future Internet promise a new ecosystem where everything would be as a service, according to a custom composition and with dynamic management of resources at run-time. Each component has the responsibility to render a service, and needs relevant information to control the business code realising this service. In this section, we first describe the properties [23] characterising SCC, as identified in the OpenCloudware project [24]. Next, we present the structure of the SCC membrane that contains the non-functional components (Section 3.2). Finally, Section 3.3 describes the interfaces of SCC components.

3.1 Properties

In this subsection we consider additional properties that characterise Cloud service component. Those properties restrict the component specification to identify the components that characterise services. For components featuring the properties exposed below, composition and adaptation are made easier, as well as the definition of the QoS featured by the SCC.

The functional component of SCC is represented in Fig. 1. This is a service that must ensure the following properties (beyond properties of SCA/GCM/SOA):

- *Stateless.* A SCC must not keep or handle information about its state, and the computation status. If a service maintains a state in the long-term, it will lose its property of loose coupling, its availability for other (concurrent) queries, as well as its potential to scalability. To be designed in a stateless way SCCs may delegate state management to other entities. For a service to be stateless, its operations need to be designed to make stateless treatments, i.e. the treatment of an operation should not rely on information received during a previous invocation or wait for the result of an external service invocation.
- *Mutualisation.* The component is a multi-tenant service component. Multiple users may require the service at the same time. This reinforces the need for the "stateless" property required above.
- *Ubiquity.* Service components can be gathered into communities of components that are equivalent in functionality and QoS. Service components are defined equivalent if they provide the same services with the same QoS even if their algorithms are different.
- *Exposability.* The functional and non-functional description of service components is provided and allows one to build an application through a catalog or a portal.

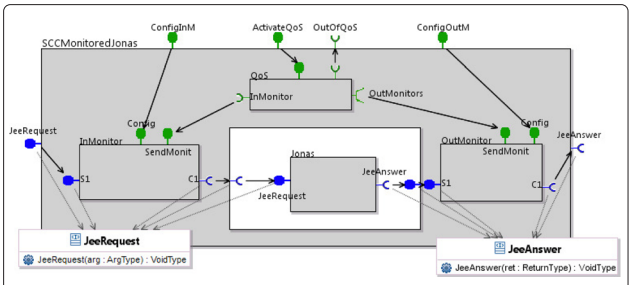


Fig. 1 Structure of a self-controlled component

These properties allow exposing components in a library (catalog), sharing components for use in different applications, and assembling them in a personalised session.

A SCC component contains:

- A *functional content* (business) with the properties defined above.
- *External interfaces* (client and server) used for communicating with the environment.
- A *membrane* for non-functional aspects that we describe in Section 3.2.

Ultimately, it is the responsability of the application architect to decide which components should be monitored, controlled, and subject to various levels of autonomicity, and at which level(s) of the application hierarchy the monitoring and control must be implemented. So the architecture proposed in this paper is based on templates (SCC, Monitors, etc.) that will be instanciated and specialized at design time.

3.2 Membrane structure

As we have seen in Section 2, in GCM components, autonomy is based on the use of a MAPE loop that we can put in the membrane of each component. But we have refined the functionality of the component to make it "as a service". Then, we have a simple and generic MAPE loop.

Indeed, we have a single server interface and identical services for all users. In this case, checking whether a component is compliant or not to a given QoS is much simplified as the component fulfills a single service provided by a single interface.

That is why we propose a "self-controlled" service component (Fig. 1 shows an instanciation of an SCC for a Jonas component) based on the triad "input monitor, output monitor, and QoS control" those different parts are described below. This contract reflects behaviour defined at design time and proposed in the offer.

We also add a property to the SCC component : "QoS offered". It will allow us to choose a service component on the basis of its behaviour (functionality and QoS).

3.2.1 Monitoring component (MaaS)

SCC have the goal to control contract compliance. This control should be based on measures taken at run-time. We propose in this paper to define monitoring components as "Monitoring as a Service" (MaaS). Monitoring components should be hosted in the membrane of each component.

MaaS components will introspect the component behaviour by taking measures. The questions that are to be answered for MaaS components are: where to measure, when to measure, and what to measure.

Where to measure? MaaS components come in three variants, depending on their position in the data flow of the application. They are observing the external behaviour (input and/or output) of the functional component inside the SCC component (see Figs 1 and 4). They play the role of interceptors: e.g. for an input MaaS, that we call InMonitor, incoming service requests are intercepted, the interceptor component stores the non-functional information about the requests, which are then transmitted (unchanged) to the functional component, via the corresponding internal interfaces. Similarly, the OutMonitor intercepts outgoing service requests. The last case is called InOutMonitor, and is used in the case of SCC components delivering a return value on their service interface.

The use of two monitors or of a two-way monitor, will give us precise numeric values in input and output of the service. Monitor components are not responsible for metric analysis, and do not take decisions. It is the QoSControl component that will make the necessary calculations to evaluate the behaviour of the service component.

When to measure? As a consequence of the MaaS placement, we will be able to take measures upon each request arrival and request emission.

What to measure? We basically propose a generic monitor that measures the number of arriving requests, the number of erroneous or rejected requests, as well as the arrival and exit time. To obtain these values we need "Counters" and "Timestamp" using the system time. More advanced monitoring functionalities could be designed to check for example the nature of the request.

3.2.2 Contract compliance component (QoSControl)

The QoSControl component is associated with the business component. It ensures compliance with the service contract.

QoS criteria To describe the behaviour of our components and permit homogeneous QoS management, we define a generic QoS model [25].

Four criteria are proposed to describe the QoS: availability, integrity, time, and capacity.

- Availability represents the accessibility rate of the service component.
- Integrity represents the capacity to run without alteration of information (for example: error rate).
- Time represents the time required for request processing (for example: response time).
- Capacity represents the maximum load the service component can handle (for example: processing capacity).

This revealed to be useful and sufficient in all the practical cases we studied.

To support the self-management of resources, for each QoS criterion, we define three values: the *design value* has been determined during the design of the service, the *current value* is the value monitored during the service lifetime, the *threshold value* represents the limit the criterion should not exceed for the component to ensures the correct processing of requests.

The QoSControl process The QoSControl component checks the current behaviour of the resource and its conformity with the contract. For this, it compares each current value to the corresponding threshold value not to exceed. It sends an OutContract notification if the current value is less (or more) than the threshold value; in this case the dynamic management consists in replacing the failing component by an ubiquitous service (see Section 6.2) fulfilling the requirements. Otherwise, it sends an InContract notification. In practice, contracts may be a bit more complicated that this, and combine checks on the values of several criteria. In Section 4.2 we show how we

specify the behaviour of the management components (monitors and controllers) and use these specifications for verification purposes.

3.3 Interfaces

We have identified 3 types of interfaces necessary to perform the self-controlled function of our SCC component: the functional, management, and control interfaces.

- Each SCC component has exactly two *functional interfaces* (`JeeRequest` and `JeeAnswer`, in blue, in Fig. 1). One server interface includes the processing functions (service methods) that can be performed by the service component. One client interface performs invocation on the next service of the chain, transmitting its current result for further treatment or for exploitation of the final result. Note that the functions of these interfaces have no return value, according to the SCC specification.
- The *management interfaces* are non-functional server interfaces (`ConfigInM, ConfigOutM, and ActivateQoS` in green in Fig. 1). They contain the necessary mechanisms to manage the configuration of non-functional components in the membrane.
- The *control interface* is a client non-functional interface (`OutOfQoS`, in green, in Fig. 1). It contains mechanisms for conveying the self-control information to the *Manager* in charge of processing QoS violation events. It outputs InContract notifications as long as the behaviour conforms to the contract, otherwise it triggers OutContract. Absence of InContract events can be used by the manager to detect severe failures from the SCC component.

The structuring logic of the membrane allows the reusability and genericity of our components. The triad (InMonitor, OutMonitor, QoSControl) associated with each component service introduces a homogeneous service component management.

4 Modelling/design platform

In this section we introduce the VerCors platform that we use for the modelling and behaviour analysis of our self-controlled components. Having a tool-supported methodology is first important for the design phase, when the designer builds his/her application, using legacy functional components as basic bricks, and assembling them following our SCC generic components. The toolsuite is also very useful for generating executable code containing the whole architecture description and the skeleton of the final application. We present the VerCors platform in Section 4.1, then discuss how to use this tool-set for formal verification of the application behaviours in Section 4.2.

4.1 The VerCors modelling tool-set

VerCors [26] is a platform for the specification, analysis, verification, and validation of GCM-based applications. The principle of the tool is illustrated in Fig. 2. First, a user specifies the architecture of a GCM-based application, the signature of its interfaces, and the behaviour of the primitive components using VerCors Component Editor (VCE). Here a first validation is performed, concerning all structural coherency aspects of the application model. This check guarantees static validity of the model, and ensures also that the code generation will terminate correctly, and that the generated code will not fail during deployment of the application components. Then, from these descriptions the behavioural model of an application is generated (ADL2N) in a form of a dedicated behavioural semantic model: a parameterised networks of synchronised automata (pNets). This semantic model is transformed by abstraction functions (ABS*), until reaching a finite model suitable for model-checking. Finally, the Model Checker, in our case the CADP tool [27], verifies the correctness of the model with respect to a set of temporal logic properties (user requirements) and in the case errors are detected it provides their description.

Once the requirements have been proven correct on the VCE specification, the user can generate the set of files allowing the deployment of the application (Architecture Description Language (ADL) for the Architecture Description, IDL for the Interface definition in the form of Java interfaces). Then naturally, the user has to provide java classes implementing the service methods of the primitive components. These files are processed by the GCM component factory to build an executable application that is executed within the GCM/ProActive execution environment.

4.2 Behaviour specification and verification

In order to generate a behavioural model in the VerCors platform, and ultimately ensure by model-checking that the overall component assembly behaves as expected (e.g. does not deadlock), the application architect should provide: 1) an architectural description, in terms of a

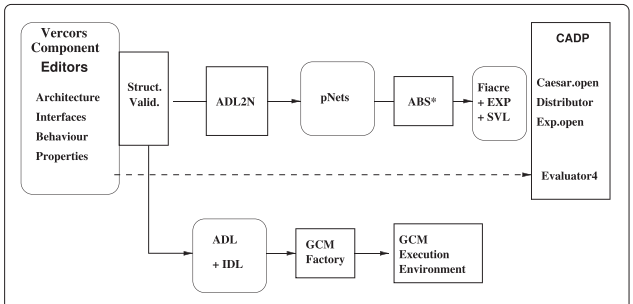


Fig. 2 Principles of the VerCors platform

composition of components, with their interfaces and bindings; and 2) for each service method of a primitive components, an abstract specification of its behaviour, e.g. in the form of a Unified Modeling Language (UML) state-machine, encoding the data- and control-flow of the component assembly. The transitions of the state-machines represent the communication events between components, including whatever data that is significant for the modelling. There is no need at this step in the design of the application to have too much detail on the real implementation; in particular any information that is only relevant to the functional code of the components should be abstracted away. The point here is that formal verification (model-checking) is performed by essence on an abstract model, not on the real executed code. Indeed, too much details would make model-checking difficult, if not impossible. The heart of the approach (but this is far from the topic of this article) is to include in the abstract model, and here in our state-machines, only what is needed to represent the behaviour of the application, including asynchronous communication, but also internal workflow of the service methods. The corresponding part of the code will be generated, and should not be modified by the developper, when adding the functional part of the code.

In Fig. 3, we give an example of the (simplified) behaviour specification of our QoSControl component. On activation, the QoSControl component has received its contract as a parameter, that we represent here as `"thr"`, the threshold values of the set of criteria. On a periodic basis, it will query the In and Out monitors for their respective criteria values, using the `getCritValues` method of the respective `InMonitor` and `OutMonitor` interfaces, and compare them with the thresholds (using the `IsInContract` predicate). The resulting diagnostic (InContract or OutContract) is sent on the non-functional interface `OutOfQoS`.

The composition of all the behaviours, together with the standard controllers of GCM, will allow the verification engine to build the full state-space of the application, and check its behavioural requirements. This construction makes use of the behavioural semantics of GCM, described in [28].

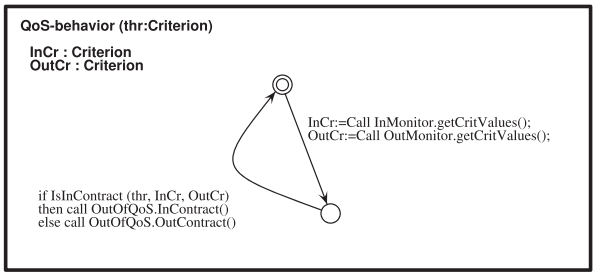


Fig. 3 Behaviour of a QoSControl component

5 Application design with autonomic control

In this section we discuss the different possibilities that our methodology offers to the application designer, when assembling components from (Cloud) services to build an autonomic application obeying SLA constraints. This means chosing which components should be SCC or not, how to organize and compose SCCs in the hierarchy, where to place the autonomic intelligence, and how to relate the monitored and controlled components with the autonomic management.

5.1 Composing SCC components

In Section 3, we have defined a GCM architecture for a standard SCC component, in which monitors observe activity on the server and client interfaces of the component. The SCC structure has many advantages for building flexible cloud service-based applications, in which services are stateless, thus easily mutualizable, replicable and exchangeable.

Most of the elements in an application will be SCC components, composed hierarchicaly using the usual GCM structure. Not all components in the hierarchy need to be monitored and controlled; both primitive components (encapsulating one single primitive service) and composite components can be controlled, and imbricated components can also be controlled simultaneously at several level of the hierarchy. In case of QoS violation, the control loop of the application will decide at which level(s) the problem should be corrected. It is the responsability of the application designer to decide which components should be controlled, and designed using the SCC template.

5.2 Using non-SCC components

There are cases where a component cannot use the basic SCC definition from Section 3. This is the case typically when this component is waiting for the result of some remote computation, before being able to finish its own task. For this kind of service, the structure of the basic SCC is not suitable: the monitoring of the service must be done between the arrival of the client request on the server interface, and the return of the result to the client, back through the same interface, when the computation is achieved. For these cases, we have a specific template with a single `InOutMonitor` inserted at the server interface, and a `QoSControl` component slightly different, taking all its information on one single client non-functional (NF) interface bound to the `InOutMonitor`, and controlling the requested QoS. An example is shown in Fig. 4.

Remark that such services with returned values should also be mutualisable, so they have to memorise the `ids` of the client requests, and use them later to return the result at the corresponding address. So they are stateful in some sense, and specific care will have to be taken if they need to be replaced. As a consequence, to build applications as

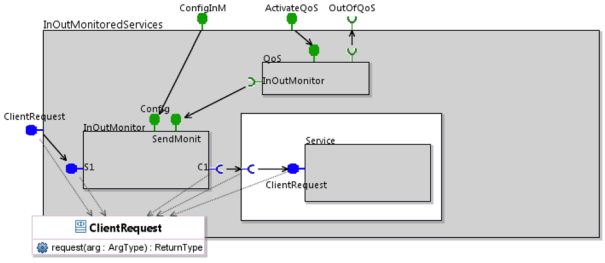


Fig. 4 Monitored component: version for services with returned values

much "as a service" as possible, meaning stateless, mutualisable and ubiquitous, the architect should be careful to separate the return value management part as much as possible, and to stick to the basic SCC model for all other components.

Last, we have to decide where the intelligence of the autonomic control will be. In principle, there is no problem having autonomic control distributed at several places and at several levels in an application: this may even be more efficient, for very large and distributed systems. However, in many cases, it will be sufficient to have one single component, at the toplevel of the application, gathering all of the QoS information, and providing the autonomic management. This mechanism is detailed in the following section.

5.3 Global autonomic control: the MAPE loop

One of the most common ways to provide autonomic behaviour is to rely on a *feedback control loop* that follows four canonical activities: *collect*, *analyse*, *decide*, and *act*. This loop defines four phases: *Monitor*, *Analyse*, *Plan*, and *Execute* and it is usually referred to as the *MAPE* autonomic control loop. In GCM, we propose a component architecture that can be embedded in the component membrane and ease the programming of autonomic adaptation procedures. These MAPE loops can either act at the top level of the composition, but also interact through the hierarchical nature of GCM applications.

We will see in the next section that SCCs are composed into compound services at a certain level, called the VPSN. It is at this level that the autonomic adaptation will be performed through the use of QoS management components in the membrane; it is at this level that the SLA will be defined and guaranteed. However, interaction with lower-level services part of the composition is still necessary, at least to monitor the services and determine the service that should be changed in order to guarantee the agreed quality of service. For this reason each service encapsulated into a component is equipped with monitoring interfaces and reports its status to the compound service. The intelligence of the adaptation is placed in the analyser of the compound service, that, based on global monitoring of its performance, on monitoring information of the sub-services, and on global additional requirements, will decide the instances of the sub-services that must be part of the composition, possibly changing the set of involved sub-services to adapt to execution settings and to contract requirements.

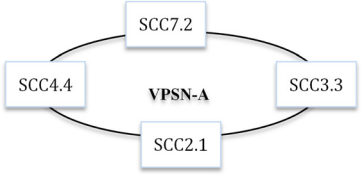
6 Service composition management

The service composition includes different service components invoked during the user session. In a context where these components are SCCs, and in order to use service components adapted to the user demand (functionality and QoS), we propose to preselect them at the session initialisation. The pre-selection (pre-provisioning) is made according to the user required QoS mapped to the service offered QoS. This service composition constitutes the VPSN (Section 6.1). Once the VPSN is built, during exploitation phase, we need to manage the composed service. We are defining three kinds of management reactions according to the decision level: operational decisions (Section 6.2), tactical decisions (Section 6.3), and strategic decisions (Section 6.4).

6.1 Service composition: the VPSN

Based on SCC service components, upon establishment of the user's session, a private service composition (the VPSN) is constructed by plugging together SCC components according to the functionality and the QoS required by the user (Fig. 5). For each VPSN, a table is created in the knowledge base; it contains the VPSN-ID, SCC-ID, their addresses and their offered QoS [29].

Because SCCs have the recommended properties (Stateless, Ubiquity, and Mutualisation), they can take part in multiple VPSNs simultaneously (shared by different users). For example in Fig. 6, SCC3.3 component is attached to VPSN-A, VPSN-B, and VPSN-C.



VPSN Table			
VPSN-ID	SCC-ID	Offered QoS	@SCC
VPSN-A	SCC2.1	QoS_SCC2.1	@SCC2.1
	SCC3.3	QoS_SCC3.3	@SCC3.3
	SCC4.4	QoS_SCC4.4	@SCC4.4
	SCC7.2	QoS_SCC7.2	@SCC7.2

Fig. 5 VPSN Table

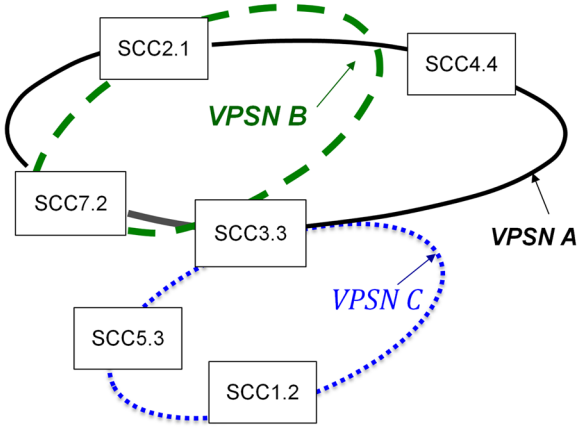


Fig. 6 Service Mutualisation and VPSN

In the next subsection, we will focus on the management based on information reported by the various monitors.

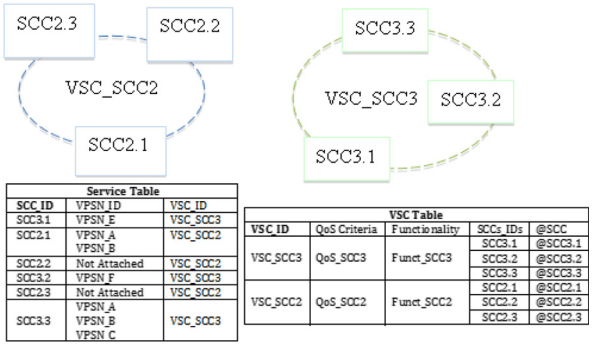
6.2 Operational decisions : VSCs and SCCs on the VPSN

The QoS control integrated to each SCC service component allows us to manage contract violations (OutContract) when they occur: it takes the appropriate operational decisions. We advocate a substitution with an SCC ubiquitous service component. The component replacement is managed by using Virtual Service Communities (VSCs). A VSC is a community of components containing several equivalent SCC components (having the same functionality and the same QoS).

The community of interest concept (VSC) allows us to react dynamically to any contract violation, to compute the adequate changes to be undertaken, and then to apply the changes in the VPSN. First, we describe the VSC creation (Section 6.2.1), and then we explain the dynamic reaction of SCCs on the VPSN (Section 6.2.2).

6.2.1 VSC creation

VSCs are created during service components deployment. When a service component is deployed, the information related to this service is saved in a Service Table. This information contains the SCC-ID, Functionality and QoS criteria. According to this created Service Table, the SCC is attached to the VSC corresponding to it in terms of QoS and functionality. Adding the SCC-ID in the VSC Table and adding the VSC-ID in the Service Table makes this attachment (Fig. 7). In the case where no VSC corresponding to this SCC, it creates a new VSC to the deployed service; which correspond to a VSC Table with following information VSC-ID, QoS criteria, Functionality, SCCs-IDs and SCC address. The VSCs can be grouped per strategic location, per operators or per platforms in order to fulfill user's requests. After explaining



Service Table

SCC_ID	VPSN_ID	VSC_ID
SCC3.1	VPSN_E	VSC_SCC3
SCC2.1	VPSN_A VPSN_B	VSC_SCC2
SCC2.2	Not Attached	VSC_SCC2
SCC3.2	VPSN_F	VSC_SCC3
SCC2.3	Not Attached	VSC_SCC2
SCC3.3	VPSN_A VPSN_B VPSN_C	VSC_SCC3

VSC Table

VSC_ID	QoS Criteria	Functionality	SCCs_IDs	@SCC
VSC_SCC3	QoS_SCC3	Funct_SCC3	SCC3.1	@SCC3.1
			SCC3.2	@SCC3.2
			SCC3.3	@SCC3.3
VSC_SCC2	QoS_SCC2	Funct_SCC2	SCC2.1	@SCC2.1
			SCC2.2	@SCC2.2
			SCC2.3	@SCC2.3

Fig. 7 Service and VSC tables

the creation phase of VSCs, we move to the management phase.

6.2.2 Dynamic reaction of SCCs on the VPSN

Each platform will have its VPSN Management. The VPSN Management has three main actions: Replace, Add or Remove Component. These actions will be solicited according to the decision rules set by the platform of the provider.

During service exploitation, when an SCC component has a change in one or more of its four QoS criterion (Reliability, Availability, Capacity and Time) it sends an OutContract. The VPSN Management receives and reacts to this notification according to the changed QoS criterion.

For example, if the OutContract results from an error rate exceeding the threshold or if the component is unavailable, then the VPSN Management replaces the component in all VPSNs to which it is attached. In our example SCC2.1 is replaced by SCC2.2 in the VPSN-A (see Fig. 8). The SCC2.1 component is also part of VPSN-B (see Fig. 6), so it will be replaced by an ubiquitous one (SCC2.2). If the OutContract is the result of a low capacity compared to the demand then the VPSN Management

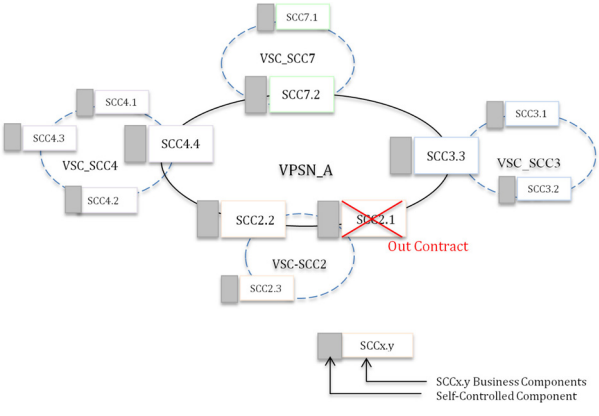


Fig. 8 SCC and VSC dynamic reaction on the VPSN

will add an ubiquitous component to cover the demand. In case where the load of traffic is decreased, the VPSN Management will remove the added component in the concerned VPSN. If the OutContract results from a higher response time than the demand, then we will have the option to add a component or to replace it by another with better response time in the concerned VPSN.

6.3 Tactical decision: MAPE loop

At the top of the composition, the management decisions are taken by the MAPE loop. These decisions are not at the operational level and as they depend on available resources, they rather are at tactical level.

Indeed, Cloud computing technology now allows customers to use cloud services according to a pay-as-you-go style. Answering a user request, together with its associated QoS, the cloud provider infrastructure proceeds to the corresponding allocation of non-shareable components adding or reducing the number of components in the service session (VPSN) according to the required capacity. These decision are in general depending on the business rules which are defined in the knowledge base; this is why, we consider them at tactical level.

This may be implemented using a software architecture with a multiple level autonomic management, in which the MAPE loops of all service compositions will interact with some shared controllers managed by the service providers. The VPSN management is obtained by the combination of the local actions of the MAPE loop components, with the external resource Managements. When some corrective action is required within a self-controlled application, the Analysis component of its MAPE loop receives the notification (OutContract) from some QoSControl components, and sends the diagnostic to the Planning component. This one will request either a replacement SCC component, or some additional resources, to the external management environment (through the VPSN- management interface). When receiving back the requested resources, it will build a reconguration script, and pass it to Execute. In Section 7, we will show an example of such an addition.

6.4 Strategic decision

Beyond these actions that can be automated, we have all the other actions of the FCAPS (Fault, Configuration, Accounting, Performance, Security) model that handle the overall management, for example, the configuration and activation actions of the QoS components, which generally are initialized by the architect according to SLA. FCAPS are standards of Telecommunications Management Network and framework for network management [30]. Finally, the rules that govern the whole are called high level and are of strategic nature. Fig. 9 shows the complementarity of automated management actions:

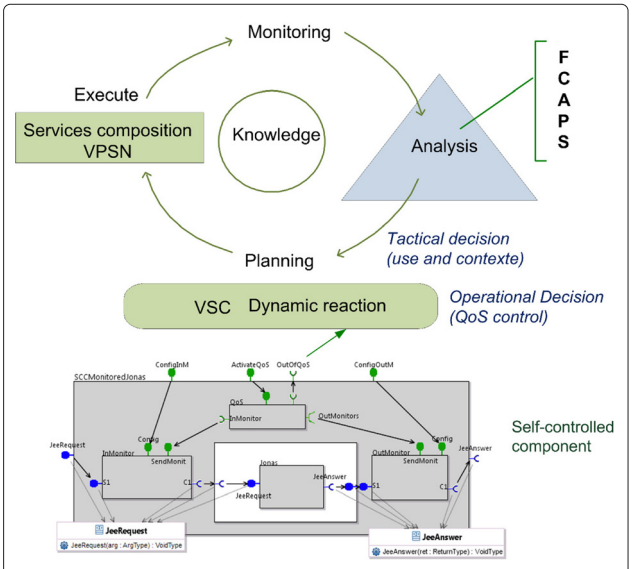


Fig. 9 Composition management

operational, tactical, and strategic. Operational decision corresponds to dynamic reaction VSC. Tactical decision is realised in the MAPE loop based on application of usage and context. Strategic decision corresponds to high level management decisions.

7 The Springoo use-case

In this section we apply the previously defined model to a use case: a Springoo application. Springoo is a web application that conforms to the three-tier Java Enterprise Edition (J2EE) platform architecture, to provide typical online merchant applications using Apache/Jonas/MySQL components. Originally based on the container framework Spring [31] it was developed as an internal test-bed for innovative technologies by Orange R&D. This application is one of the end-to-end case-studies of the OpenCloudware project for Cloud development and management infrastructures. We use it in this paper to show the advantages of self control and SLA management.

Our new proposal for the architecture of (a subset of) Springoo is shown in Fig. 10; note that to improve the readability of this diagram, we have hidden many labels (in particular interface names), and most of the UML interface specifications. To implement this application, our partners proposed a PaaS which can be described as an open platform for cloud software engineering, accessible to cloud architects and developers, deployed on multi-IaaS through a self-service portal.

Our work has allowed us to define Springoo using mostly SCC components, with their In- and Out- monitors and their QoSControl component. We have defined the architecture described below.

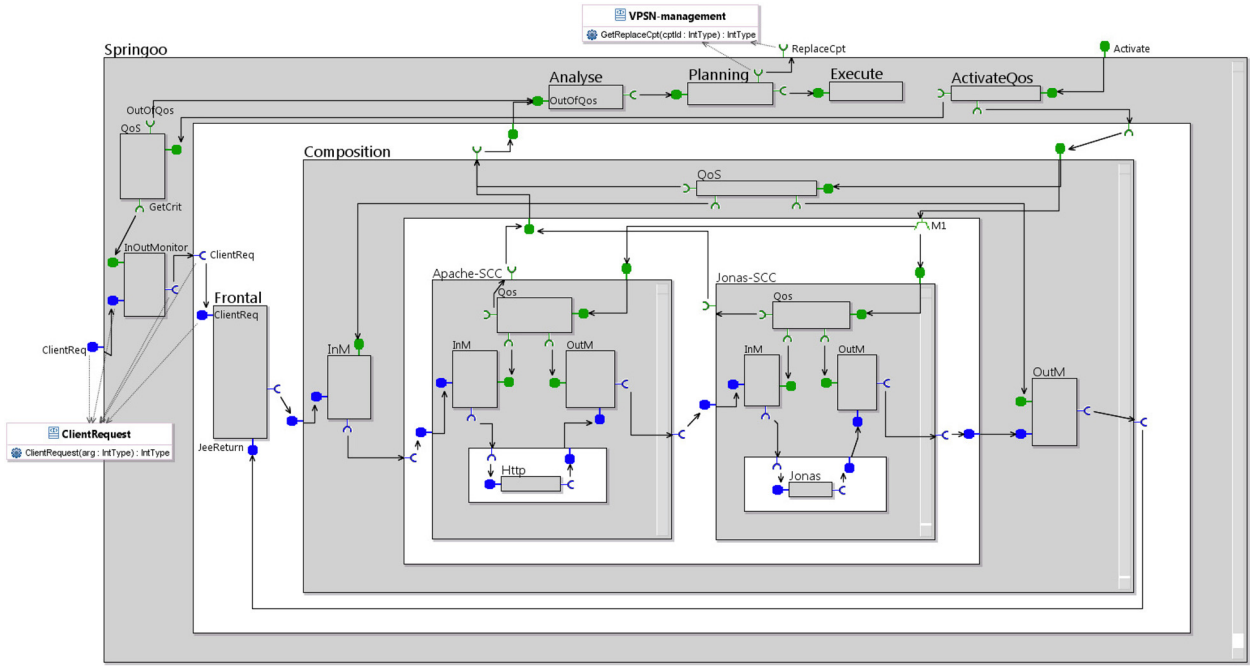


Fig. 10 Architecture of the Springoo application

First, all the main components of Springoo (Composition, Apache-SCC and Jonas-SCC) are generically defined. They are all of the self-control and stateless type, as described in Section 3.2. To keep the example simple enough, we do not describe here the details of the Jonas component. Recall that an SCC component includes an input monitor, an output monitor and a service quality control component (QoSControl). Springoo is therefore composed of a set of primitive and composite components, all of SCC type.

The only exception is the frontal component, that receives the client requests, transmits them to Apache, and waits for a return value from Jonas. The frontal awaits for answers, either from Apache if the Http components detect an error (typically when the query syntax is incorrect - code 400 - not shown in the figure for lack of space), or from Jonas when the request is fully processed. In both cases, the frontal returns the request answer (error or response value) to the client, through the InOutMonitor of the Springoo component. As a consequence, the `ClientRequest` service method, on the external Springoo service interface, but also on the Frontal service interface, has a significant `ReturnType`. But all the other service methods (HttpRequest, JeeRequest, HttpReturn) have a Void return type.

The Apache service has been designed as a pure SCC component named Apache-SCC, with a Http subcomponent (in the real world, it would also have an Https subcomponent, and these would be used to process the client requests depending on the protocol used (http or https). The Apache-SCC server interface `HttpRequest`, and the Http component server interface have a single service method, that receives the http request as an argument, and do not return any value. After successful decoding of the http request, they send a JeeRequest (through their OutMonitor components, abbreviated OutM in Fig. 10) to the Jonas-SCC component.

The Http and Jonas components are monitored and controlled at their own level, but also at composition level if this is necessary according to SLA. If an OutContract event is raised on the non-functional interface of Composition and Apache-SCC and Jonas-SCC are InContract, this would mean that their link is faulty. In the case where the three notifications are OutContract, the manager will have the choice to manage the problem by replacing the sub-components in fault, or the whole composition.

Next, a full MAPE loop has been integrated in the membrane of the Springoo component. Five internal components define this loop:

- An input/output monitor, and the associated "QoSControl" component, as described in Section 5, analyze the client requests and responses ("InOutMonitor").
- An "Analyse" component receives all the indications of violation of QoS contracts for sub-components. Depending on the situation, it may take 2 corrective decisions: either a replacement of a component by a

similar one matching the contract; or an increase in the resources allocated to the failing service.

- A "Planning" component receives the diagnostic from the Analyse, and plans the actions to be performed. Depending the type of correction, it will either request a replacement SCC component, or some additional resources, to the external management environment (through the VPSN-management interface). When receiving back the requested resources, it will build a reconfiguration script, and pass it to Execute.
- An "Execute" component is responsible for executing the required reconfiguration. When receiving the new SCC components from the environment, it will transmit them to the adequate place in the distributed hierarchy, and launch the execution of the reconfiguration scripts.

Additionally, we have an "Activate", component, responsible for enabling or disabling the service quality control of all QoSControl components in the hierarchy.

Finally, 3 interfaces have been defined to communicate with the outside:

1. ClientReq: An input/output bidirectional use interface responsible for processing the client requests and responses.
2. ReplaceCpt: A client interface informing the external Manager components that a sub-component needs to be replaced or added.
3. Activate: A server interface which allow all QoSControl components to be enabled or disabled.

Elasticity and Load balancing Adding the standard notion of elasticity in our architecture is not very difficult: when the MAPE loop has identified the need to augment the computing capacity by adding a component (and the corresponding resources) to share the workload, it has signaled this request to the external *Manager* through the `VPSN-management` interface. As an answer, the Manager will select an adequate component (from the corresponding VPSN), and return it to Springoo. We illustrate this in Fig. 11, in which we want to manage elasticity for the Jonas service, using a set of Jonas instances, and an autonomous management of this set using a MAPE loop and the VPSN management mechanisms. To process the dynamic reconfigurations, we have replaced the single Jonas Qos-aware component by a Qos-aware composite component, containing a `LoadBalancer` component, and the Jonas instances. The LoadBalancer has a multicast client interface `M1` that scatters the Jee requests on the Jonas replicas. The Springoo toplevel MAPE loop receives information from the composite Jonas monitors, and decides whether it needs to adapt (add or remove) the set of replicas. Suppose for example the planning component decides to augment the capacity, it sends an AddCpt request on the VPSN-management interface, gets back the id of a new Jonas replica, and generates a GCM script that will be executed at the level of the LoadBalancer.

A similar mechanism can of course be used for any component of the application (Http, Https, and Jonas sub-components), at the Springoo toplevel, or even

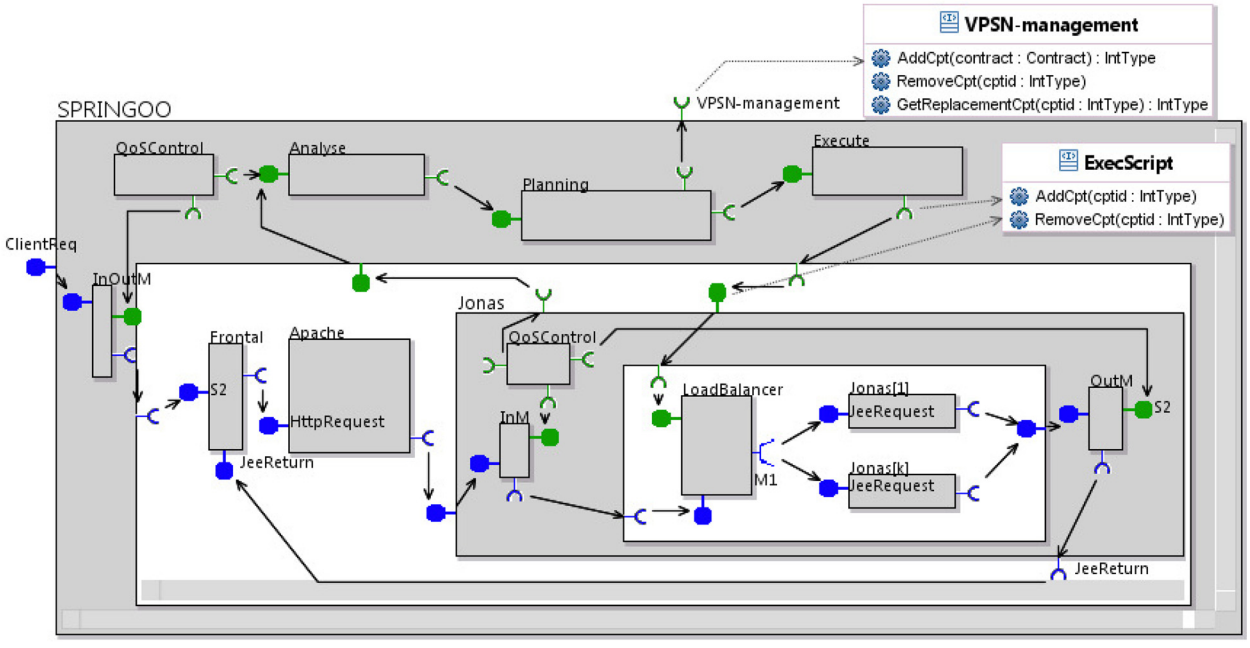


Fig. 11 Load balancer

combined at several of the hierarchy, if the architect wants it.

This structure has been specified, and validated by the VCE Editor. Here "validated" means structural validity, staticaly checked by VCE, including correctness of the architecture, interfaces, and namings (full definitions in [32]). This guarantees that the generated ADL will be valid, and the application will be built without runtime error by the component factory. The generated ADL code gives the description of the whole Springoo Application. It encompasses functional and non-functional components, associated classes, interfaces, and bindings.

8 From SCC component to SLA

This section complements our contributions to the non-functional aspects through a QoS model. It presents the generic SLA (Service Level Agreement) model that we have proposed in the OpenCloudware project and as an ETSI standard [33]. The specificity of this model is the following: it explicits on the one hand the SLO user requirements, and on the other hand the provider offers (service and QoS) associated to the same QoS model (four generic criteria). To meet the SLO requested by the user, the provider offer will rely the composition of several SCC components.

First, in this section, we begin with the description of our SLA approach (Section 8.1). Next, we present (Section 8.2) the SLO expression in adequacy with the management actions provided by SCC components. Finally, in Section 8.3, its SLA generic model.

8.1 Approach

Our approach is to propose a SLA model that explicits and aligns user requirements (SLO and Obligation) and provider offers (services) through the model and the QoS expression [34]. As we have seen in Section 3, a service (SCC component) is defined by its function and its behaviour, described according to four QoS criteria with information on QoSCriteriaName (Availability, Integrity, Time, Capacity) and QoSCriteriaValue (design, threshold and current value). The user will express his/her SLO through the four criteria. The provider selects in his/her catalog the SCC component to best meet the SLO request.

The personalised service delivery is performed according to QoS of VPSN user session. The VPSN guarantee is taken into account in SLA. For each VPSN we introduces a set of services necessary to control and manage the end-to-end QoS. Thanks to the MAPE loop of the SCC components, the service level management is automatised. The service delivery takes into account the user profile, the context, and SLO expressed by the user.

8.2 Service level objective

The Service Level Objective (Fig. 12, bullet 3.1) is the means for the user to express his needs. The objectives expressed by the user may be linked to the end-to-end QoS. The end-to-end (E2E) objectives define the final service QoS level (compound service) provided to the user. Indeed, if the customer requires a service composition, he may then precise the conditions linked to their operation such as response time, availability rate and scalability.

For Springoo example, the user requires that the processing time of his service is less than 2s in 90 % of the cases, if the number of requests processed is less than 1000 req/s. On the other hand, if the number of requests is between 1000 req/s and 2000 req/s, he may still require a processing time less than 2s but with a rate of failure less stringent (for example 80 % instead of 90 % of the cases).

The SLO linked to this user needs are then the followings:

- SLO1: E2E processing time < 2s if the request number is <1000 req/s in 90 % of the cases.
- SLO2: E2E processing time < 2s if the request number is >1000 and <2000 req/s in 80 % of the cases.

8.3 Service level agreement

The SLA is a document, a contract, that defines the specific and personalised deal, accord required between a service provider and a client [35]. After having introduced the SLA content, the Fig. 12 formalises the SLA template generic model based on UML. The SLA template generic model is composed by:

1. Parties (Signatory Party and Third Parties).
2. High level constraints
3. User part, corresponding to the demand: Service Level Objective and coverage.
4. Provider part, corresponding to the offer: Services offered as well as the associated QoS. Each service is a SCC component.
5. The SLA contract conditions.

To clarify our SLA generic model, we take the use case Springoo presented in Chapter 7.

The Parties represent the contracting entities of a SLA contract (Fig. 12, bullet 1). We classify these entities as "Signatory parties" and "Third Parties". The first set represents the contractual parties that can include the provider, the end-user, the developer (Springoo resource requester), etc. The second set represents the trusted third parties involved in the SLA contract including the network provider, the monitoring provider, etc. Not applicable to the Springoo.

The proposed SLA model includes the conditions imposed by the provider and/or the consumer high level

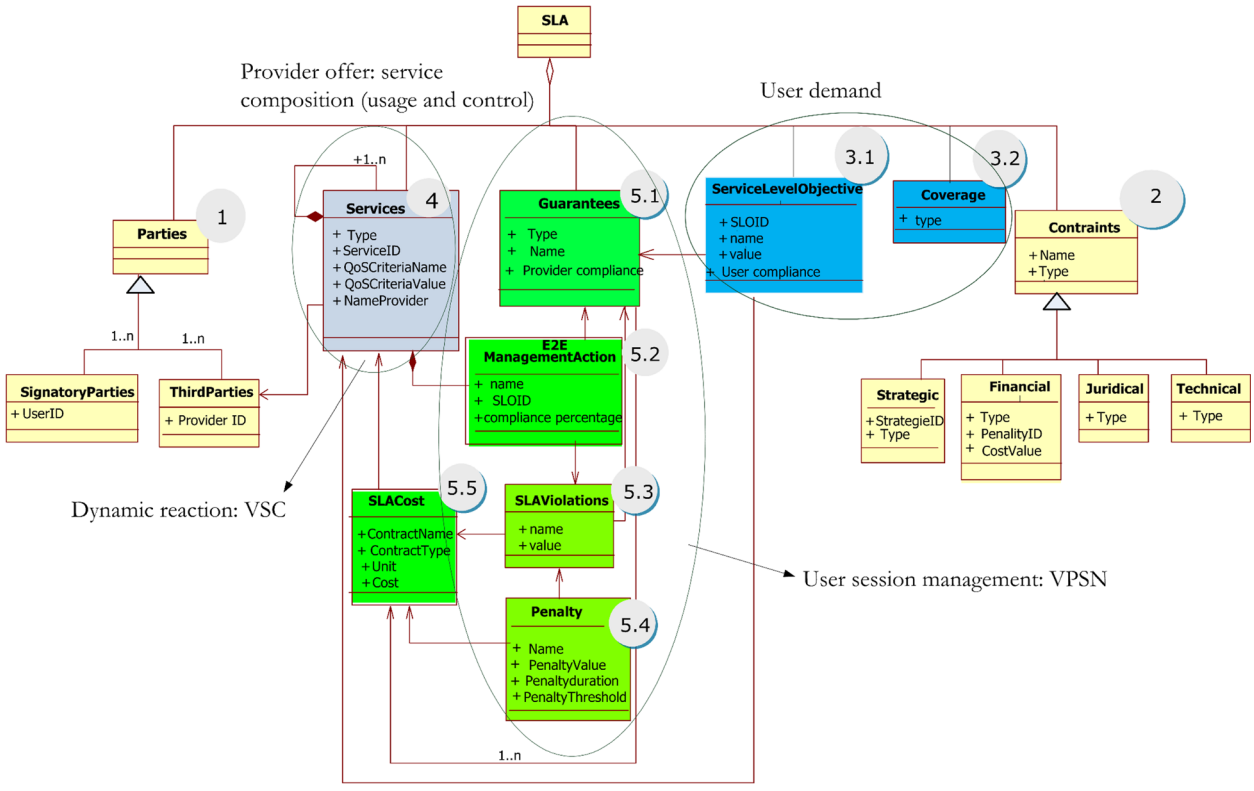


Fig. 12 SLA generic model

constraints (Fig. 12, bullet 2). We propose the following classification of constraints: strategic, financial, juridical, and technical. Strategic constraints represent either the strategies requested by the user or those applied by the provider. Financial constraints are the conditions related to the payment and usage patterns. Juridical constraints define legal constraints such as licensing rules, editing rights, etc. Technical constraints represent the requirements determined by the provider to execute the requested service. For example, the provider may require the user to have a specific browser for Springoo.

The user part (Fig. 12, bullet 3) should be composed of SLO, geographical features and coverage.

The offer of the provider represents the catalog of components available on the PaaS. The providers offer SCC components that are differentiated by their QoS levels, their prices and how they are built, deployed and managed. In a specific SLA (Fig. 12, bullet 4) we find the result of the choice of SCC components corresponding to the SLO of the user and constitute its VPSN (SCC composition). In our Springoo example, the selected SCCs are an Apache-SCC and a Jonas-SCC component whose composition meets the requested SLO (number of requests per second, response time and integrity 8.2). The classes (Fig. 12, bullet 5) represent the terms of contracts. We have the supplier guarantees namely SLO1 and SLO2 of the user of Springoo (Fig. 12, bullet 5.1). To do this, we find the "SLA Management Actions" (Fig. 12, bullet 5.2), which define the list of actions to do, more those provided at the SCC components level (6), ie at the VPSN level, for the end-to-end service delivery. We find the actions that are induced by SLO and those for SLA violation. For example, to ensure SLO2 of the user, which represents resources elasticity case, according to requirements, we need to add a Jonas and therefore a Load Balancer component to share the load. These shares will be automatically activated when an OutContract of QoS component at the composition level will be sent. Another case would be that of SLA violations (Fig. 12, bullet 5.3), for example, if the response time of the VPSN no longer meets the SLO. The specific actions will depend on the business rules and diagnosis. If a violation response time comes from the location of the added Jonas, it must be replaced with an ubiquitous Jonas with a more appropriate location. These management actions are conducted by the dynamic reconfiguration of the VPSN.

The penalties (Fig. 12, bullet 5.4) define the benefits to the user, in case of SLA violation. The SLA costs (Fig. 12, bullet 5.5) cover all costs associated with the signed SLA and those of management actions taken.

9 Experiments

We have partially implemented[1] the architecture from Fig. 10 using the GCM/ProActive middleware to provide the Monitors, QoSControl, and MAPE components. GCM/ProActive comes with all the software infrastructure necessary to simply define interceptor components and easily integrate monitoring components in the composition. The implementation provides a MAPE loop that monitors the response time of both Apache and Jonas components and is able to trigger QoS decisions on them. Two criteria are implemented: (1) enable/disable properties like caching on Apache components, which can improve the response time, but may be more costly in storage; (2) whenever the average response time of the Jonas component increases over T secs, the QoSControl component notifies it through the OutOfContract interface; then, the MAPE triggers an `AddCpt` action through the `ExecScript` interface, to add a Jonas worker inside the Jonas component. Jonas uses a `LoadBalancer` component as the one shown in Fig. 11, to distribute the load through the workers, as described in Section 7.

Figure 13 shows three stages of the service. In the first stage there is only one Jonas component serving requests, and an SLO specifies that the average response time of Springoo must be less than 4 secs. After the first 20 requests, an OutContract notification is sent from Apache and the MAPE loop decides to enable caching on the HTTPS component, which is slower than HTTP; this results in a decrease in response time. During the second stage, an SLO in the Jonas component specifies that the average response time must be less than $T = 1.8$ secs. When the OutContract is sent by Jonas, the MAPE loop decides to add a new Jonas component to share the load of the application server. After 40 requests it can be seen that the SCC Springoo component has been able to decrease its response time guided by the SLOs and autonomic actions.

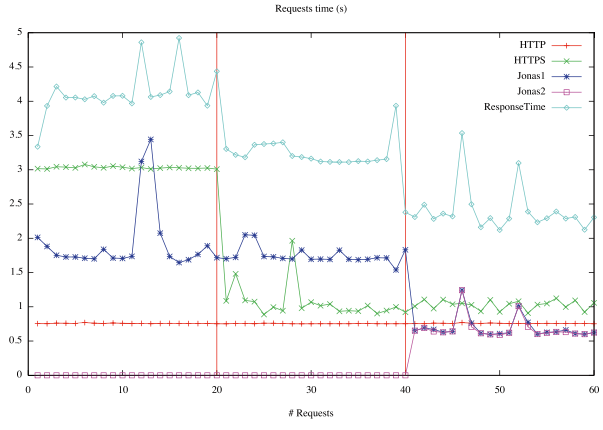


Fig. 13 Request response time

10 Conclusion

The main contribution of this paper is to unify the concepts of components and services in the context of cloud applications, so that the various promises of the new service ecosystem become a reality.

We clarified the concept of service through properties so that the component "as a service" becomes an entity of service composition. The aim is to choose and to assemble application services into a network of services that can be loosely coupled to create flexible and dynamic processes (VPSN). This new composition paradigm allows the personalised design of complex applications that automatically adapt to a required service level agreement, possibly by changing the services involved in the composition.

We show in this paper with an "Apache as a Service" use-case named Springoo, how the adoption of self-controlled components helps the service composition to provide a guarantee of quality of service.

The approach we advocate covers the whole spectrum from architectural modelling, to service implementation, and run-time support including autonomic contract management (SLA).

Our proposal is backed-up by a design and verification platform, used to build early models of the applications, check their properties, and generate code supported by an execution environment: GCM/proactive. These environments were used in the implementation of a Springoo scenario that shows the feasibility of our proposals.

Endnote

[1]This implementation is available at: https://github.com/scale-proactive/mape-component-controllers/tree/qosaware. It contains only a small part of our QoS-aware model of Springoo, tailored to demonstrate the QoS autonomic management features.

Abbreviations

ADL: Architecture Description Language (section 3.2.2); GCM: the Grid Component Model (section 2.1); MaaS: Monitor as a Service (section 3.2.1); MAPE: Monitor-Analyse-Planning-Execute (loop) (section 5.3); SCC: Self-Controlled service Component (section 3); SLA: Service Level Agreement (section 8); SLO: Service Level Objective (section 8); SOA: Service Oriented Architecture (section 2.1); QoS: Quality of Service (section 3.1); VPSN: Virtual Private Service Network (section 6.1); VSC: Virtual Service Community (section 6.2); SCA: Service Component Architecture (section 2.1).

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

All authors are equal contributors. All authors read and approved the final manuscript.

Authors' information

- **Tatianna Aubonnet** is assistant professor at computer science department of CNAM (Paris). She holds a PhD in computer and network science from Telecom ParisTech. Her research interests cover service creation and management in Next Generation Networks.

- **Ludovic Henrio** is researcher at CNRS, I3S laboratory. His research expertise includes programming languages, distributed systems, formal methods, and component-oriented programming.
- **Soumia Kessal** has a PhD thesis from Telecom ParisTech, his research interests are about Cloud Services Management from Deployment to Delivery, Service Design and QoS Management.
- **Oleksandra Kulankhina** is a PhD Student at INRIA Sophia Antipolis and University of Nice-Sophia Antipolis. Her research interests include distributed systems, component-oriented programming and model-driven engineering.
- **Frédéric Lemoine** has an engineering degree in computer science, microelectronics and automatics. He is a research engineer and development project manager at the computer science department of CNAM. His expertise includes programming and modelling languages, heterogeneous parallel systems programming, embedded systems and mobile devices programming.
- **Eric Madelaine** is a senior researcher at INRIA in Sophia-Antipolis, France. He has an engineer diploma from Ecole Polytechnique, Paris, a PhD in computer science from Univ. of Paris 7, and an Habilitation from Univ. of Nice. His research interests range from semantics of programming languages, distributed and cloud computing, component-based software, formal methods, methods and tools for specification and verification of complex programs.
- **Cristian Ruz** has a PhD from University of Nice-Sophia Antipolis, and currently works as a researcher at Pontificia Universidad Católica de Chile. His research interests include distributed and autonomic computing, software components and high performance computing.
- **Noëmie Simoni** is an Emeritus Professor of Telecom-Paristech. She was Head of Architecture and Engineering of Networks and Services (AIRS) research group at the Department of Computer Science and Network. Her research interests include QoS management and modelling of complex systems. Her expertise, gained through many academic projects and industrial contracts, covers wide range of management topics. Today, her main work is focused on network convergence and service convergence, network virtualisation and cloud computing.

Acknowledgment
This work is supported by the OpenCloudware project. OpenCloudware is funded by the French FSN (Fond national pour la Société Numérique), and is supported by Pôles Minalogic, Systematic and SCS.

Author details
[1]CNAM (Conservatoire National des Arts et Métiers), Computer Science Department, 292 rue Saint Martin, 75003 Paris, France. [2]CNRS, University of Nice Sophia-Antipolis, Sophia-Antipolis, France. [3]Telecom ParisTech, 46 rue Barrault, 75013 Paris, France. [4]Inria, Sophia-Antipolis, France. [5]Pontificia Universidad Católica de Chile, Santiago de Chili, Chile.

Received: 23 January 2015 Accepted: 3 July 2015
Published online: 25 July 2015

References

1. Bruneton E, Coupaye T, Leclercq M, Quéma V, Stefani JB (2006) The Fractal component model and its support in Java. Softw: Prac Exp 36(11–12):1257–1284. doi:10.1002/spe.767
2. Baude F, Caromel D, Dalmasso C, Danelutto M, Getov V, Henrio L, Přez C (2009) GCM: a grid extension to Fractal for autonomous distributed components. Ann Telecommun 64(1–2):5–24. doi:10.1007/s12243-008-0068-8
3. Baude F, Henrio L, Ruz C (2015) Programming distributed and adaptable autonomous components - the GCM/ProActive framework. Softw Prac Exp. doi:10.1002/spe2270
4. ProActive Parallel Suite. http://proactive.inria.fr/
5. Service Component Architecture Specifications (2007). http://www.opengroup.org/standards/soa
6. Guinard D, Trifa V, Karnouskos S, Spiess P, Savio D (2010) Interacting with the SOA-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. IEEE Trans Serv Comput 3:223–235. doi:10.1109/TSC.2010.3
7. Zhao H, Doshi P (2009) A hierarchical framework for logical composition of web services. Serv Oriented Comput Appl 3(4):285–306. doi:10.1007/s11761-009-0052-9
8. Lawton G (2008) New ways to build rich internet applications. IEEE Computer 41(8):10–12
9. Liu X, Hui Y, Sun S, Liang H (2007) Towards service composition based on mashup. In: IEEE SCW. IEEE Computer Society. pp 332–339. http://dblp.uni-trier.de/db/conf/IEEEscc/scw2007.html
10. Hung SH, Shieh JP, Lee CP (2011) Migrating android applications to the cloud. Int J Grid High Perform Comput 3(2):14–28. doi:10.4018/jghpc.2011040102
11. Zhang Q, Cheng L, Boutaba R (2010) Cloud computing: state-of-the-art and research challenges. J Internet Serv Appl 1(1):7–18
12. Lenk A, Klems M, Nimis J, Tai S, Sandholm T (2009) What's inside the cloud? an architectural map of the cloud landscape. In: Software Engineering Challenges of Cloud Computing, ICSE Workshop on. pp 23–31. doi:10.1109/CLOUD.2009.5071529
13. Boutrous-Saab C, Coulibaly D, Haddad S, Melliti T, Moreaux P, Rampacek S (2009) An integrated framework for web services orchestration. Int J Web Service Res 6(4):1–29
14. Seinturier L, Merle P, Rouvoy R, Romero D, Schiavoni V, Stefani JB (2012) A component-based middleware platform for reconfigurable service-oriented architectures. Softw: Prac Exp 42(5):559–583. doi:10.1002/spe.1077
15. Hnětynka P, Plášil F (2006) Dynamic reconfiguration and access to services in hierarchical component models. In: Proceedings of the 9th International Conference on Component-Based Software Engineering. CBSE'06. Springer. pp 352–359. doi:10.1007/11783565_27. http://dx.doi.org/10.1007/11783565_27
16. Parashar M, Liu H, Matossian V, Schmidt C, Zhang G, Hariri S (2006) Automate: Enabling autonomic applications on the grid. Cluster Comput 9:161–174. doi:10.1007/s10586-006-7561-5
17. David PC, Ledoux T (2006) An aspect-oriented approach for developing self-adaptive Fractal components. In: Löwe W, Südholt M (eds). Software Composition. LNCS. Springer, Berlin Heidelberg Vol. 4089. pp 82–97. doi:10.1007/11821946_6
18. IBM (2006) An architectural blueprint for autonomic computing. white paper Fourth Edition. http://citeseerx.ist.psu.edu/viewdoc/download?, doi:10.1.1.150.1011&rep=rep1&type=pdf
19. Gauvrit G, Daubert E, Andre F (2010) Safdis: A framework to bring self-adaptability to service-based distributed applications. In: Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications. SEAA '10. IEEE Computer Society, Washington, DC, USA. pp 211–218. doi:10.1109/SEAA.2010.25
20. Lango J (2014) Toward software-defined slas. Commun ACM 57(1):54–60. doi:10.1145/2541883.2541894
21. Van HN, Tran FD, Menaud J (2009) Sla-aware virtual resource management for cloud infrastructures. In: Ninth IEEE International Conference on Computer and Information Technology, Xiamen, China, CIT 2009, 11-14 October 2009, Proceedings, Volume I. pp 357–362. doi:10.1109/CIT.2009.109. http://dx.doi.org/10.1109/CIT.2009.109
22. Wada H, Champrasert P, Suzuki J, Oba K (2008) Multiobjective optimization of sla-aware service composition. In: 2008 IEEE Congress on Services, Part I, SERVICES I 2008, Honolulu, Hawaii, USA, July 6–11, 2008. pp 368–375. doi:10.1109/SERVICES-1.2008.77. http://dx.doi.org/10.1109/SERVICES-1.2008.77
23. Aubonnet T, Simoni N (2014) Self-control cloud services. In: 2014 IEEE 13th International Symposium on Network Computing and Applications, NCA 2014, Cambridge, MA, USA, 21–23 August, 2014. pp 282–286. doi:10.1109/NCA.2014.48
24. The OpenCloudware project. http://www.opencloudware.org/
25. Aubonnet T, Simoni N (2013) Service creation and self-management mechanisms for mobile cloud computing. In: Wired/Wireless Internet Communication - 11th International Conference, WWIC 2013, St. Petersburg, Russia. Proceedings. pp 43–55. doi:10.1007/978-3-642-38401-1_4
26. Cansado A, Madelaine E (2009) FMCO 2008. LNCS. In: de Boer FS, Bonsangue MM, Madelaine E (eds). Springer, Berlin Heidelberg Vol. 5751. pp 180–203. doi:10.1007/978-3-642-04167-9_10
27. Garavel H, Lang F, Mateescu R, Serve W (2011) Cadp 2010: A toolbox for the construction and analysis of distributed processes. In: TACAS'11.

LNCS, vol. 6605. Springer, Saarbrücken, Germany. doi:10.1007/978-3-642-19835-9_33
28. Ameur-Boulifa R, Henrio L, Madelaine E, Savu A (2012) Behavioural Semantics for Asynchronous Components. Rapport de recherche RR-8167, INRIA. http://hal.inria.fr/hal-00761073
29. Soulimani HA, Coude P, Simoni N (2011) User-centric and qos-based service session. In: 2011 IEEE Asia-Pacific Services Computing Conference, APSCC 2011, Jeju, Korea (South), December 12–15, 2011. pp 267–274. doi:10.1109/APSCC.2011.64. http://dx.doi.org/10.1109/APSCC.2011.64
30. TMN: M.3400 TMN management functions. Technical report, ITU-T (2000). Standard, https://www.itu.int/rec/T-REC-M.3400/en
31. The Spring framework. http://www.spring.io
32. Henrio L, Kulankhina O, Liu D, Madelaine E (2014) Verifying the correct composition of distributed components: Formalisation and Tool. In: FOCLASA, Rome, Italy. https://hal.inria.fr/hal-01055370
33. ETSI EG 202 009-3, V1.2.1: User group; quality of telecom services; part 3: Template for service level agreements (sla). Technical report, European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2014). standard. http://webapp.etsi.org/WorkProgram/Report_WorkItem.asp?WKI_ID=19268
34. Ayadi I, Simoni N, Aubonnet T (2013) Sla approach for "cloud as a service". In: Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing. IEEE Computer Society, Washington, DC, USA. pp 966–967. doi:10.1109/CLOUD.2013.127. http://dx.doi.org/10.1109/CLOUD.2013.127
35. TMF GB917: SLA management handbook; release 2.5. Technical report, TeleManagement Forum (TMF) (2011). Standard, https://www.tmforum.org/resources/standard/gb917-sla-management-handbook-volume-2-concepts-and-principles-release-2-5-zip/